Cybersecurity

RESEARCH Open Access

Feedback control can make data structure layout randomization more cost-effective under zero-day attacks

Ping Chen[1*], Zhisheng Hu[2], Jun Xu[1], Minghui Zhu[2] and Peng Liu[1]

Abstract

In the wake of the research community gaining deep understanding about control-hijacking attacks, data-oriented attacks have emerged. Among data-oriented attacks, data structure manipulation attack (DSMA) is a major category. Pioneering research was conducted and shows that DSMA is able to circumvent the most effective defenses against control-hijacking attacks — DEP, ASLR and CFI. Up to this day, only two defense techniques have demonstrated their effectiveness: Data Flow Integrity (DFI) and Data Structure Layout Randomization (DSLR). However, DFI has high performance overhead, and dynamic DSLR has two main limitations. **L-1:** Randomizing a large set of data structures will significantly affect the performance. **L-2:** To be practical, only a fixed sub-set of data structures are randomized. In the case that the data structures targeted by an attack are not covered, dynamic DSLR is essentially noneffective. To address these two limitations, we propose a novel technique, *feedback-control-based adaptive DSLR* and build a system named SALADSPlus. SALADSPlus seeks to optimize the trade-off between security and cost through feedback control. Using a novel feedback-control-based adaptive algorithm extended from the Upper Confidence Bound (UCB) algorithm, the defender (controller) uses the feedbacks (cost-effectiveness) from previous randomization cycles to adaptively choose the set of data structures to randomize (the next action). Different from dynamic DSLR, the set of randomized data structures are adaptively changed based on the feedbacks. To obtain the feedbacks, SALADSPlus inserts canary in each data structure at the time of compilation. We have implemented SALADSPlus based on gcc-4.5.0. Experimental results show that the runtime overheads are 1.8%, 3.7%, and 5.3% when the randomization cycles are selected as 10s, 5s, and 1s respectively.

Keywords: Data structure manipulation attack, Data structure layout randomization, Adaptive security, Feedback control

Introduction

During the past two decades, control-hijacking attacks have drawn tremendous attention from the computer security research community. In a control-hijacking attack, the adversary manipulates the control flow objects and shifts the execution to malicious logics. The earliest attacks hijack the control flow to execute injected code. To defend against those code-injection attacks, Data Execution Prevention (DEP) (The PaX Team 2003a; Microsoft 2008) techniques were proposed. DEP ensures that a memory page is either writable or executable, but not both.

As a counteraction against DEP, adversaries switched from code-injection attacks to code-reuse attacks such as return-to-libc and Return-Oriented-Programming (ROP). These code-reuse attacks have motivated a very large amount of research on how to defend and how to counter-attack. In the past 10 years, the research community has gained deep understanding about the cost-effectiveness of major defenses, including Address Space Layout Randomization (ASLR)[1] (Backes and Nürnberger 2014; Bhatkar et al. 2003; Kil et al. 2006; Keromytis et al. 2012; The PaX Team 2003b) and Control Flow Integrity (CFI)[2] (Abadi et al. 2005).

*Correspondence: pzc10@ist.psu.edu; chenping19851@hotmail.com
[1]College of Information Sciences and Technology, The Pennsylvania State University, University Park 16802, PA, USA
Full list of author information is available at the end of the article

However, in the wake of the research community gaining deep understanding about control-hijacking attacks, data-oriented attacks (Chen et al. 2015, 2005; Hu et al. 2015, 2016) have emerged. Data-oriented attacks do not modify control flow objects. Instead they read/write security-sensitive data objects for malicious goals (Chen et al. 2005; Hu et al. 2015). Recently, it has been shown that data-oriented attacks are Turing-complete (Hu et al. 2016) and can result in arbitrary behaviors.

Among data-oriented attacks, data structure manipulation attack (DSMA) (Chen et al. 2015) is a major category. DSMA exploits memory corruption bugs to manipulate multiple security sensitive fields in encapsulated data objects (e.g., `struct` and `class`). For example, the attack against `openssh` (CVE-2001-0144 2001) (CVE-2001-0144) overwrites a particular instance of data structure `passwd` to achieve privilege escalation. Pioneering research was conducted and shows that DSMA is able to circumvent the most effective defenses against control-hijacking attacks — DEP, ASLR and CFI. However, the research community has quite limited understanding on how to defend against DSMA.

Up to this day, only two defense techniques have demonstrated their effectiveness: Data Flow Integrity (DFI) (Castro et al. 2006; Song et al. 2016) and Data Structure Layout Randomization (DSLR) (Chen et al. 2015; Lin et al. 2009; Stanley et al. 2013; Xin et al. 2010). DFI maintains the definition-use relationship from the Data Flow Graph, and checks whether the definition of each data object is legal at run-time. By theory, DFI can defend against DSMA. However, DFI introduces performance overhead as high as 103% (Castro et al. 2006), making it impractical for deployment. Comparing with DFI, DSLR has similar defense effectiveness but substantially less cost. We, therefore, believe DSLR is much more promising in mitigating DSMA. In this work, we seek to provide new insights into and deeper understanding about the cost-effectiveness of DSLR.

DSLR

Research in the early stage proposed static DSLR (Lin et al. 2009; Stanley et al. 2013; Xin et al. 2010). At the time of compilation, static DSLR randomly reorders the fields or adds dummy fields in encapsulated data objects. Static DSLR can prevent DSMA from correctly locating target fields and further manipulating them. However, its randomization is fixed at runtime and vulnerable to brute force attacks. Further, static DSLR requires manual efforts to determine which data structures can be randomized.

Recent research endeavored to develop dynamic DSLR and produced a technique named SALADS (Chen et al. 2015). SALADS aims to address the limitations of static DSLR. It automatically determines the randomization-feasibility of each data structure and frequently re-randomizes/de-randomizes the layouts of candidate data structures at run-time. The program compiled by SALADS can self-randomize a set of of data structures, the instrumentation replaces each statement that contains data structure accesses. To avoid runtime errors, SALADS inserts de-randomization routine before any dangerous statement (e.g., pointer involved dangerous statements).

While SALADS offers security advantages, it still has two major limitations. **L-1:** When SALADS randomizes a large set of data structures, it will significantly affect the performance. This further leads to the second limitation. **L-2:** For the consideration of performance, SALADS cannot afford to randomize all the data structures. Instead it randomizes a fixed sub-set of data structures. In the case that the data structures targeted by an attack are not covered, SALADS is essentially noneffective.

Problem statement

In this paper, we explore to augment SALADS with feedback control to address the above limitations. Our insights are as follows.

Limitation L-1 essentially indicates the necessity of a trade-off between security and cost. The availability of feedbacks about security and cost will facilitate the defense to achieve an optimized trade-off. This motivates us to employ the canary mechanism to collect feedbacks. More details about it will be discussed shortly. Those feedbacks in turn provide awareness of the attacked data structures, which can be leveraged to address limitation L-2. The intuition behind is that such awareness can enable the defense to include the attacked data structures and exclude the safe ones for randomization.

The goal to optimize the above described trade-off can be formulated as a feedback control problem — The defender (controller) uses the feedbacks (cost-effectiveness) from previous randomization cycles to adaptively choose the set of data structures to randomize (the next action) such that the trade-off is optimized.

In this paper, we are therefore deeply interested in the problem: *Can feedback control be leveraged to address the two limitations of dynamic DSLR?*

Our approach

Based on the above insights, we propose a novel technique, *feedback-control-based adaptive DSLR* and build a system named SALADSPlus. SALADSPlus includes two parts. The first part is a compiler extension, which transforms a program into a Data Structure Self-Randomizing (DSSR) program. The compiler extension mounts the DSSR program into an adaptive strategy that utilizes the Upper Confidence Bound DSLR (UCB-D) algorithm to select data structures for protection at each re-randomization. Different from SALADS, the set of randomized data structures are adaptively changed based on the cost-effectiveness utilities of the previous cycles. To obtain those utilities, SALADSPlus inserts canary in each data structure at the time of compilation. At runtime and the end of each defense cycle, SALADSPlus collects the number of polluted canaries. Such information is then used to calculate the utilities.

The second part of our SALADSPlus system is an adversarial reasoning scheme. This scheme monitors the execution of a DSSR program and uses the observations to reduce the uncertainty in detecting ongoing DSMA. In addition, this scheme can help locate the program targeted by the DSMA.

Our contributions in this work are summarized as follows:

- This is the first effort toward feedback-control-based adaptive DSLR.
- A novel feedback-control-based adaptive defense algorithm extended from UCB (Upper Confidence Bound) algorithm (Auer et al. 2002) is proposed.
- An adversarial reasoning scheme is proposed. It enables the defender to know more about the attacker. It also helps locate the program targeted by the DSMA.
- On average, the runtime overheads introduced by SALADSPlus are 1.8%, 3.7%, and 5.3% for application programs (`SPECInt2006`, `openssh-2.1.1p4`, `httpd-1.1.1`, and `openssl-0.9.6d`), when the defense cycles are 10s, 5s, and 1s, respectively.

Overview

Our work focuses on developing a specific defense against DSMA (Chen et al. 2015), named SALADSPlus, before any patches are generated and the zero-day memory corruption bugs are located. This section presents the motivation and overview of our defense system. Under the protection of a defense action, the server uses canary detection to identify the number of failed DSMAs during a defense cycle and reports the number to the defense decision maker ("Canary detection" section). The feedback is used by the defense decision maker to calculate the cost-effectiveness of previous defense action ("Cost-effectiveness utility" section), and select defense action for next defense cycle (i.e., a data structure whitelist in "Dynamic data structure layout randomization" section).

Feedbacks and adaptive defense

During the vulnerability window, the defender has limited knowledge of the attacks; e.g., the bug locations, target data structures, etc. Under this limited knowledge, it is not realistic for the defender to eliminate all attacks. Therefore the defense goal is to increase the difficulty for the attacker to succeed with low cost. As discussed, static DSLR is fixed at runtime and vulnerable to brute force attacks. And dynamic DSLR techniques (e.g., SALADS) have to face the trade-off between security and cost. If a large set of data structures are re-randomized, it will significantly affect the performance. On the other hand, if the data structures targeted by the attacker are not covered, it cannot provide security. To achieve an optimized trade-off, we employ the canary mechanism to collect feedbacks and periodically select a (small) set of data structures to randomize at runtime based on the observable feedbacks. In particular, SALADSPlus utilizes memory forensic approach to find the polluted canaries. The polluted canaries are sent as the feedbacks to the defender. We will show in "Evaluation" section that the polluted canaries reflect how many DSMAs are blocked (failed DSMAs) and thus can be used to evaluate the effectiveness of a defense action. Those feedbacks can provide awareness of the attacked data structures, which can be leveraged to include the attacked data structures and exclude the safe ones for randomization.

The key idea of SALADSPlus is trial-and-error learning. More specifically, SALADSPlus receives polluted canaries, which reflect how many DSMAs are blocked and evaluate how well the deployed actions are. On one hand, SALADSPlus selects the optimal actions that can block most DSMAs in the history (exploitation), on the other hand, SALADSPlus also tires seemingly non-optimal actions choices (exploration). So SALADSPlus is particularly well suited to defend against DSMAs where the defender is unaware of the targets of DSMAs but can evaluate its previous actions via repeated interactions with attackers. The details of how the defender updates its actions by utilizing the feedbacks will be illustrated in "Cost-effectiveness utility and UCB-D algorithm" sections. And the unique capability of the SALADSPlus, adversarial reasoning, will be introduced in "Adversarial reasoning scheme" section and evaluated along with the cost-effectiveness in "Evaluation" section. The uniform time interval between two consecutive updates of the defense actions is denoted as a defense cycle and the whole vulnerability window is denoted as N defense cycles; i.e., $T \triangleq \{1, 2, \cdots, N\}$. Note that we do not specify the feedbacks and the update frequency of the attacker. In the rest of the paper, we evaluate the security from the point of view of the defender.

System components

In the system model depicted in Fig. 1, we consider two entities: the defender and the attacker. The attacker launches the DSMA. In order to circumvent the defense, the attacker uses brute force attacks to locate the target

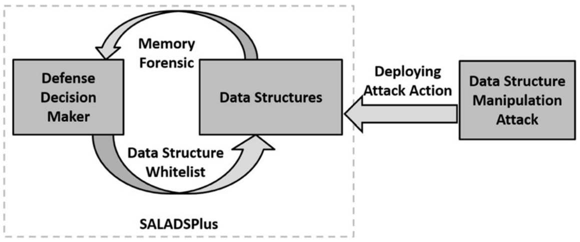

Fig. 1 Overview of how DSSR Binary defends against DSMA

data structures. The defender, named SALADSPlus, consists of two components: *defense decision maker* and *defense actions.*

Data Structure Manipulation Attack (DSMA) In this paper, we consider one attacker with multiple DSMAs. In general, an attacker launching DSMAs is associated with a set of attack actions denoted by $\mathcal{A} \triangleq \{a_1, \cdots, a_m\}$, where each action is a combination of several attack scripts. One attack script targets several data structure types (once a script is fixed, the targets are fixed).

Data Structure Self-Randomization Binary (DSSR Binary) SALADSPlus uses adaptive DSLR with adversarial reasoning and generates Data Structure Self-Randomization binary (DSSR Binary). The DSSR binary maintains the metadata for all the data structure instances, including the base addresses and relative positions of the fields in data structures. In addition, all the data structure read/write operations are replaced with a set of DSSR statements (Chen et al. 2015) to access the randomized data structure layout. What's more, all the definitions of the data structures are randomized at compile-time, and padding bytes are inserted into the data structures (Lin et al. 2009). The DSSR binary is equipped with a set of defense actions denoted by $\mathcal{D} \triangleq \{d_1, \cdots, d_n\}$, where each defense action randomizes a particular set of data structures at runtime. The defense decision maker adaptively updates its actions. The adaptive update rule will be briefly discussed in Section Feedbacks and adaptive defense. In order to get the feedbacks for the defender, canary detection inserts 32-bit specific values for each field into data structures of a DSSR binary. Once DSSR binary detects a polluted canary, it indicates an attacker has maliciously modified the fields in a data structure; reports the program name and the target data structure to the security officer.

Design and implementation

In this section, we first present dynamic data structure layout randomization, and then demonstrate how the canary detection generates cost-effectiveness utility values. Further we illustrate the cost-effectiveness utility value and the UCB-D algorithm.

Dynamic data structure layout randomization

As discussed in Section ref, dynamic DSLR techniques (e.g., SALADS) are facing two limitations.Our solution SALADSplus uses a novel UCB-D algorithm to dynamically make decisions on "which data structures to randomize", while SALADS sticks to a fixed set of data structures. To solve the first limitation, we develop an adaptive algorithm (in "UCB-D algorithm" section) to dynamically decide "which set of data structures to randomize during the next defense cycle" based on the cost-effectiveness utility value (in "Cost-effectiveness utility" section) of the previous decisions. The algorithm enables the defender to choose better actions as time goes by. To reduce the overhead, SALADSPlus re-randomizes data structures much less frequently (i.e., once per defense cycle) but in a much more adaptive certain sense (more adaptivity in general provides more resilience to DSMA).

At the beginning of each defense cycle, DSSR binary randomizes the data structures in the dynamic whitelist, and de-randomizes the data structures that are not in the dynamic whitelist. There are two challenges when we design SALADSPlus: (1) changing the dynamic whitelist of data structures at runtime without recompiling the program (SALADS compiles the source code with a static whitelist); (2) when DSSR binary randomizes data structures, multiple DSSR statements are executed based on the previous layout of the data structures. Without concurrency methods, the DSSR statements may access totally irrelevant fields.

Dynamic WhiteList To solve the first challenge, DSSR binary maintains a dynamic whitelist. A dynamic whitelist is a buffer which is allocated in the heap and inserted by our customized compiler. It consists of the indices of the data structures. If a data structure is in the whitelist, DSSR binary will randomize it in a defense cycle. Otherwise, the data structure will keep its pre-known layout. The only code of DSSR binary that can access the dynamic whitelist is an independent thread illustrated as follows.

Independent Thread Independent thread is a thread which is inserted by our customized compiler at the entry of DSSR binary. The thread will allocate the dynamic whitelist and update the whitelist periodically. The thread calls the UCB-D function, which we will present shortly in "UCB-D algorithm" section, to "know" exactly how to update the whitelist. Once the whitelist is updated, the thread will de-randomize the data structures in the previous whitelist, and randomize the data structures in current whitelist. In addition, the thread contains the canary detection (in "Canary detection" section).

Write Preferring Lock To solve the second challenge; i.e., avoiding the inconsistency between the DSSR statements and the independent thread, we propose a write preferring lock method. We create a mutex lock and a global counter which calculates how many DSSR statements are currently executed. The thread will require a lock before it randomizes/de-randomizes the data structures, and release the lock after it finishes the randomization/de-randomization. Before the thread does the randomization/de-randomization, it waits until all the concurrent DSSR statements are completed; i.e., the global counter is 0. DSSR statements will check the lock before accessing the data structure. If the lock is occupied by the thread, the statements will wait. When DSSR statements are executed, they will firstly increase the global counter by 1, and after accessing the data structure, the global counter decreases by 1.

Canary detection

The canary detection scheme generates cost-effectiveness utility values. In particular, the canary detection is executed in the thread at the end of each defense cycle based on the memory forensic analysis. It scans the canaries in the randomized data structures and compares current values with a random canary value (Crispin et al. 1998). The random canary is chosen at the beginning of each defense cycle through `/dev/urandom`. If a canary in a data structure is polluted, we regard it as one failed DSMA. To quickly pinpoint the canary, we maintain an array to record the addresses of the canaries and mark each element as `0` or `1`, where `1` indicates that the canary needs to be checked. After DSSR statements complete the data structure access, the canary detection checks whether the data structure type is in the whitelist, and then DSSR statements will update the array for the corresponding canaries.

Cost-effectiveness utility

The UCB-D algorithm is a utility-based reinforcement learning algorithm. As mentioned in "Feedbacks and adaptive defense" section, the key idea of SALADSPlus is to utilize feedbacks generated by the server in history to evaluate the corresponding defense actions and gradually identify the optimal actions. We define *utility* to quantify the cost-effectiveness of a defense action. In particular, the utility is in the form of $u = W_r r - W_c c$, where r is the effectiveness and c is the cost. And the constant weights W_r and W_c are chosen according to the preference of the defender on security and efficiency.

The most straightforward quantification of the effectiveness is the number of failed DSMAs during a defense cycle. It is mentioned in "Canary detection" section that if a canary in a data structure is polluted, we regard it as one failed DSMA. Therefore, we use the number of polluted canaries to quantify the effectiveness in each defense cycle. Note that the number of polluted canaries is determined by both the attack action (the combination of attack scripts) and defense action (randomized data structures). Then the effectiveness can be represented as a mapping from $\mathcal{A} \times \mathcal{D}$ to $\mathbb{R}$, i.e., for defense cycle t, the effectiveness is $r(t) = r(a(t), d(t))$.

If adaptive DSLR does not incur any cost, then the best defense is to randomize all feasible data structures. However the study of SALADS (Chen et al. 2015) shows that the performance overhead is proportional to the number of randomized data structures. For example, SALADS introduces 110%, 120% runtime overhead when randomizing 20% of data structures in `gzip` and `gap` respectively. We use the number of randomized data structures to quantify the cost in each defense cycle. The number of randomized data structures is only determined by the defense action. Then the cost can be represented as a mapping from $\mathcal{D}$ to $\mathbb{R}$, i.e., for defense cycle t, the cost is $c(t) = c(d(t))$.

Since the numbers of DSMAs and the numbers of data structures in all defense cycles are finite, the utility values in all defense cycles are also bounded. More formally, there are u^- and u^+ such that $\forall t \in T$, $u(t) \in [u^-, u^+]$. Additionally, the defender knows the bounds. Note that there might be some DSMAs that cannot achieve their attack goal but bypass all canaries. Therefore, the utility cannot precisely represent the cost-effectiveness of a defense action because the effectiveness part r may contain error. We introduce utility error to represent bypassing DSMAs. More formally, the utility error for each defense cycle t is denoted as $\epsilon(t)$ and $u(t) = u^*(t) - \epsilon(t)$, where $u^*(t)$ is the ground truth utility if all failed DSMAs can be detected and $\epsilon(t)$ represents the number of failed DSMAs which bypass the canary detection. Our canaries are randomized by the adaptive DSLR and thus difficult to bypass. So $\epsilon(t)$ is small. This will be validated in "Evaluation" section.

UCB-D algorithm

With the cost-effectiveness utility value, the defense problem can be formulated as: how to choose a sequence of defense actions to maximize the sum of received utility values during the vulnerability window T. For simplicity, we define the sum as the aggregate utility. The UCB-D algorithm (an extension of the UCB algorithm in Multi-armed Bandit problems (Kuleshov and Precup 2014; Lai and Robbins 1985)) is proposed to solve the problem. A set of notations will be introduced as follows before the steps of the algorithm:

- $\mathbf{1}_{\{\Pi\}}$ is an indicator function: $\mathbf{1}_{\{\Pi\}} = 1$ if Π is true and $\mathbf{1}_{\{\Pi\}} = 0$ if Π is false.

Algorithm 1 The UCB-D Algorithm

```
1:  for $d \in \mathcal{D}$ do
2:      $T_d(1) = 0$;
3:      $\bar{\mu}_d(1) = 0$;
4:  end for
5:  for $t = 1; t \leq N; t++$ do
6:      for $d \in \mathcal{D}$ do
7:          if $T_d(t) == 0$ then
8:              $I_d(t) = +\infty$;
9:          else
10:             $I_d(t) = \bar{\mu}_d(t) + (u^+ - u^-)\sqrt{\frac{2\ln(t)}{T_d(t)}}$;
11:         end if
12:     end for
13:     $d(t) = \arg\max_{d \in \mathcal{D}} I_d(t)$;
14:     $T_{d(t)}(t+1) = T_{d(t)}(t) + 1$;
15:     for $d \in \mathcal{D} \setminus \{d(t)\}$ do
16:         $T_d(t+1) = T_d(t)$
17:     end for
18:     Defender receives $u(t)$;
19:     for $d \in \mathcal{D}$ do
20:         $\bar{\mu}_d(t+1) = \frac{1}{T_d(t+1)} \sum_{\tau=1}^{t} (u(\tau)\mathbf{1}_{\{d(\tau)=d\}})$;
21:     end for
22: end for
```

- $T_d(t) = \sum_{\tau=1}^{t-1} \mathbf{1}_{\{d(\tau)=d\}}$ is the number of times defense action d has been chosen by the end of defense cycle $t-1$.
- $\forall d \in \mathcal{D}, \bar{\mu}_d(t) = \frac{1}{T_d(t)} \sum_{\tau=1}^{t-1} \left(u(\tau)\mathbf{1}_{\{d(\tau)=d\}}\right)$ represents the *empirical average utility* the defender actually receives by choosing defense action d by the end of defense cycle $t-1$.
- $\forall d \in \mathcal{D}, I_d(t) = \left(\bar{\mu}_d(t) + (u^+ - u^-)\sqrt{\frac{2\ln(t)}{T_d(t)}}\right)$ represents the *upper confidence index* of action d at the beginning of defense cycle t.

In particular, at the beginning of the defense cycle t, the defender updates $I_d(t)$ of each defense action (Line 6–11). The indices of the actions that have never been chosen are set to be far larger than others'. In this way, these actions will be chosen with higher priorities (Line 7–8). For the actions that have been chosen before, their indices are updated based on their empirical average utility values (Line 9–11). The defender chooses the new action $d(t)$ with the largest index (Line 13) and updates the numbers of times each defense action has been chosen (Lines 14–17). At the end of the defense cycle t, the defender receives utility value $u(t)$ (Line 18) and then updates the empirical average utility values of all defense actions (Line 19–20).

An attractive feature of the UCB-D algorithm is that the defender can maximize the aggregate utility value with limited information of DSMAs. In particular, the algorithm only requires the defender to know its previous actions and their induced utility values. In contrast, it does not require the defender to pinpoint the attacked data structures. In the UCB-D algorithm, the defender, on one hand, uses average utility value (the first term of $I_d(t)$) in the history predict how well an action might work in the future and selects the most successful action, and on the other hand, tries less successful actions by the penalty term (the second term of $I_d(t)$). Through the repeated interactions with the attacker, the defender gradually identifies the data structures which are more likely attacked and randomizes them more often than others.

Adversarial reasoning scheme

From the experiments in next section, we will see SALADSPlus can provide good effectiveness with low performance overhead. In this section, we discuss another capability of SALADSPlus: it can enable the security officer to do two-level adversarial reasoning in real time. First, the security officer can determine whether a zero-day attack is DSMA or not. Second, if a zero-day attack is DSMA, the security officer can do program level reasoning to infer the target program of the DSMA. This two-level adversarial reasoning is elaborated as follows.

First Level Adversarial Reasoning The basic idea of the first level adversarial reasoning is to compare a zero-day attack with some known attacks (including DSMAs and non-DSMAs) at runtime and infer whether the zero-day attack is DSMA or not. Note that the utility defined in "Cost-effectiveness utility" section can be used to quantify cost-effectiveness of our defense. Since our defense is only effective when defending against DSMAs (determined by the DSSR Binary), the aggregate utility values of DSMAs and non-DSMAs are very different. Therefore by comparing the aggregate utility of the zero-day attack with those of known attacks, the security officer can tell whether the zero-day attack is DSMA or not. The aggregate utility values of the known attacks are achieved in Matlab simulations. We simulate SALADSPlus and the known attacks in Matlab because the simulations are much faster than the real experiments in web servers. And the simulation results are similar to the real experiment results. This similarity is ensured by the following three aspects: (1) The Matlab simulations have the same features as the real-world vulnerable web servers in terms of data structure types and instances. It is difficult to simulate the whole servers in Matlab, but we simulate the data structures and the related manipulations in Matlab. (2) The same UCB-D algorithm is implemented in both the real-world web servers and the Matlab simulations. (3) The simulated attacks have the same features as known CVEs in terms of targets and attack frequencies.

Second Level Adversarial Reasoning If a zero-day attack is a DSMA, the security officer can further infer which program the DSMA is targeting. This adversarial reasoning capability is provided by the canary detection. The canary detection reports two messages when some canaries are polluted: (1) the type of polluted data structures; (2) the program name which is inserted into the DSSR programs. With the program name and data structure type, the security officer can quickly pinpoint the target program. This second level adversarial reasoning is only meaningful when SALADSPlus is effective; i.e., the zero-day attack does not succeed. For example, the attack (CVE-2002-0656 2002) lasted several days against SALADSPlus but still failed[3]. This effectiveness, which will be validated in "Evaluation" section, gives the security officer sufficient time to locate the target program of the zero-day attack. Note that the first level adversarial reasoning is important because the security officer can quickly rule out non-DSMAs and do second level adversarial reasoning.

Evaluation

In this section, we present the evaluation of SALADSPlus. We first introduce the evaluation environment in "Real-world environment" section. We then evaluate the effectiveness of SALADSPlus in "Effectiveness" section and its performance overhead in "Performance overhead" section. We finally verify the adversarial reasoning capability of SALADSPlus in "Adversarial reasoning" section.

Real-world environment

We implement SALADSPlus on the top of gcc-4.5.0 with 12K lines of C code added. All evaluation experiments are conducted on an Intel(R) Core(TM) i5 machine with 4GB memory running Red Hat Linux 7.3 with Linux kernel version 2.4.18.

Effectiveness

How DSSR applications are generated We generate DSSR applications via using SALADSPlus to compile open source programs, including `apache-1.1.1`, `openssh-2.1.1p4`, `openssl-0.9.6d`, and `glibc-2.2.2`. The DSSR applications contain 348, 47, 132, and 2329 data structure types, respectively. The following experiments are conducted on a vulnerable apache web server (apache-1.1.1 compiled with openssl-0.9.6d and glibc-2.2.2) and a vulnerable ssh server (openssh-2.1.1p4 compiled with glibc-2.2.2). For the vulnerable servers, we divide data structures to five groups, where each group has 20% data structures. We choose the length of the defense cycle as 1/5/10 seconds, respectively. At the beginning of each defence cycle, we randomize the data structures in one group based on the UCB-D algorithm.

How attacks are launched We launch six real world attacks shown in Table 1. In the first attack, the buffer overflow bug in `openssl` (CVE-2002-0656 2002) is exploited to overwrite a data structure instance `session` (of type `ssl_session_st`) and `malloc_chunk`, whose details have been presented in "Data Structure Manipulation Attack (DSMA)" section. In the second attack, the integer truncation bug in (CVE-2001-0144 2001) is exploited to overflow the `pw_uid` in `passwd` type and do privilege escalation. The third attack exploits the heap overflow bug in (CVE-2015-0235 2015), which will pollute `malloc_chunk`. In the fourth attack, the stack overflow bug in (CVE-1999-0071 1999) is exploited to overflow `timeval`. In the fifth attack, Heartbleed bug (CVE-2014-0160 2014) is exploited to over-read 2-bytes buffer and leak sensitive data. In the sixth attack, the same bug in openssh (CVE-2001-0144 2001) is used to modify an authentication flag (Chen et al. 2005) and circumvent the authentication check. This attack does not affect any data structure, and we denote it as `do_authentication` attack. Both Heartbleed and `do_authentication` attacks are non-DSMAs.

Effectiveness We compile the selected programs with static DSLR and SALADSPlus, respectively. During our experiments, we also enable ASLR in the execution environments. We compare the defense results of static DSLR and SALADSPlus by launching the six attacks respectively. Defense results are also shown in Table 1. The results demonstrate that in two hours, all six attacks can succeed when static DSLR is deployed. When SALADSPlus is deployed, DSMAs cannot succeed within two hours but the non-DSMAs; e.g., Heartbleed and `do_authenticated` attack, can succeed.

Justification of Effectiveness Part in Utility The cost-effectiveness is represented by the difference between the number of failed DSMAs (effectiveness part) and the number of randomized data structure instances (cost part). As mentioned in "Cost-effectiveness utility" section, SALADSPlus uses the canary detection to indicate the failed DSMAs. Therefore we define the number of polluted canaries as the effectiveness part in our utility. Table 2 shows that the polluted data structures detected by the canary detection can reflect failed DSMAs, which justifies the effectiveness part of our utility.

Performance overhead

Runtime Overhead To evaluate the runtime overhead introduced by SALADSPlus, we test a number of programs, including SPECInt2006, `httpd-1.1.1`,

Table 1 Defense results of DSSR applications in two hours

Programs	CVE #	Bugs	Data Structure	Static DSLR	SALADSPlus
openssl-0.9.6d	CVE-2002-0656	KEY ARG bug (CVE-2002-0656 2002)	ssl_session_st	×	√
			malloc_chunk		
glibc-2.2.2	CVE-2015-0235	GHOST bug (CVE-2015-0235 2015)	malloc_chunk	×	√
openssh-2.1.1	CVE-2001-0144	CRC-32 bug (CVE-2001-0144 2001)	passwd	×	√
apache-1.1.1	CVE-1999-0071	Cookie bug (CVE-1999-0071 1999)	timeval	×	√
openssl-1.0.1c	CVE-2014-0160	Heartbleed bug (CVE-2014-0160 2014)	N/A	×	×
openssh-2.1.1	CVE-2001-0144	do_authentication (CVE-2001-0144 2001)	N/A	×	×

`openssh-2.1.1p4` and `openssl-0.9.6d`. We insert the instrumented code to calculate the number of randomized data structure instances at runtime. Table 3 shows the results. The defense cycles are 1/5/10 seconds and in each defense cycle, 20% data structures are randomized. As Fig. 2 shown, the average runtime overheads are 5.3%, 3.7%, 1.8% on average. The performance results show that the runtime overhead is in parallel to the randomized data structure instances.

Memory Overhead We compare the memory usage of DSSR programs with original programs. As Fig. 3 shows, the memory overhead is 1.8% on average. The memory overhead is orthogonal to the defense cycle, and mainly introduced by the paddings and canaries.

Adversarial reasoning

In this section, we verify that SALADSPlus enables the the security officer to do adversarial reasoning in real time.

Simulation Settings First we simulate SALADSPlus and the same six attacks in Matlab and get their corresponding aggregate utility values. The Matlab simulations have the same features as the vulnerable apache web server and ssh server in terms of data structure types and instances. We select 5 seconds as the length of one defense cycle. And the simulated attacks have the same features as CVEs in terms of targets and attack frequency.

Validation of the Similarity First we compare the aggregate utility values in Apache and Openssh experiments with those in Matlab simulations to validate the similarity between results of the Matlab simulations and real-world web server experiments. The results are shown in Fig. 4. We can see that the aggregate utility values of both the DSMAs and non-DSMAs (Heartbleed and `do_authentication`) achieved in Matlab simulations are similar to those achieved in Apache and Openssh experiments respectively.

First Level Adversarial Reasoning We use mutations of the attacks to simulate the zero-day attacks, and then compare the aggregate utility values of the mutated attacks achieved in Apache and Openssh experiments with those of the known attacks in simulations to tell whether the zero-day attacks are DSMAs or not. We mutate the attack scripts in two ways: (1) changing the attack target; (2) merging multiple attack scripts into one. First, we change the attack script by exploiting CVE-2001-0144 (CVE-2001-0144 2001): instead of modifying `pw_uid` in `passwd`, we write an additional attack script to manipulate `pw_passwd` in `passwd`. Second, we merge the attack scripts that exploit openssl (CVE-2002-0656 2002) and apache (CVE-1999-0071 1999) into one attack at the mixing ratio of 10 to 1. Figure 5 shows that the curves of the mutated attacks are similar to original DSMAs, but different from the non-DSMA, which verify the first level adversarial reasoning.

Second Level Adversarial Reasoning For zero-day DSMAs, we infer their target programs. From the experimental results, all the four DSMAs mentioned in Section

Table 2 Justification of Effectiveness in Utility

Programs	CVE #	Bugs	# Polluted ds in 5s	# Attacks in 5s
openssl-0.9.6d	CVE-2002-0656	KEY ARG bug (CVE-2002-0656 2002)	10	10
glibc-2.2.2	CVE-2015-0235	GHOST bug (CVE-2015-0235 2015)	16	16
openssh-2.1.1	CVE-2001-0144	CRC-32 bug (CVE-2001-0144 2001)	10	10
apache-1.1.1	CVE-1999-0071	Cookie bug (CVE-1999-0071 1999)	50	50

Table 3 The number of randomized data structure instances at runtime

Programs	httpd-1.1.1	openssh-2.1.1	openssl-0.9.6d	astar	bzip2	gcc	h264ref	
# Instances	114	89	245	12	1093	689	47	
Programs	libquantum	omnetpp	sjeng	gobmk	hmmer	mcf	perlbench	specrand
# Instances	19	24	13	15	14	23	98	0

Effectiveness can be detected by the canary detection. The correct target programs' names and the polluted data structures are reported to the security officer.

Discussion

Our adaptive defense can perform adversarial reasoning to tell whether the attack is DSMA or not. However, if the attacker knows the defense, it can tailor the attack actions; e.g., extending the duration of an attack try to several defense cycles. As such, the canary detection may not be able to detect the DSMA in some defense cycles, and cost-effectiveness utility values in these defense cycles may be very close to non-DSMA. Nonetheless, the defense can still get the feedbacks from the server and adaptively update its actions. As time goes by, the defender will gradually choose better actions so the long term aggregate utility will improve and be different from that of the non-DSMA.

Our adaptive defense is deployed before any patches are generated and the zero-day memory corruption bugs are located. During the vulnerability window, the defender has limited knowledge of the attacks. Under this limited knowledge, it is not realistic for the defender to guarantee that no DSMA can succeed. However, as one form of moving target defense, our goal is to increase DSMA costs and make it harder for the attacker to succeed. In particular, a DSMA might succeed in one defense cycle. But when the same attack is launched the next time, the attacker still has to spend a very large price to succeed. In addition, a failed attack could tamper with some data and have some side effects. But due to randomization, the attacker should have no idea of the locations of the tampered data in a short period. So the probability of facilitating following attack attempts should be very small.

The canary detection can detect continuous buffer overwrite attacks, which are the main form of buffer overflow. There are several methods to bypass the canary detection (Litchfield 2003; Team C 2009). However, most of the bypass canary methods are focused on stack cookie (Team C 2009). Some methods (Litchfield 2003) even need to hook the data structures (e.g., exception handler registration structure) to bypass. In contrast, our adaptive DSLR can raise the bar for this kind of bypassing. In addition, discrete write approaches, which use bugs like format string (OWASP 2009), can modify target data objects without changing the value of the canary. As such, discrete write can circumvent the canary detection. What's more, to circumvent the canary detection, an attacker can resort to memory content leakage (e.g., memory disclosure (Snow et al. 2013), uninitialized memory tracking (Chen et al. 2011), side channel (Bittau et al. 2014; Seibert et al. 2014; Zhang et al. 2012)). At client-side, the canaries may be easier to be read than sever-side, because just-in-time compilers (e.g., Javascript and Actionscript engines) may help the attacker to poke around memory (Blazakis 2010). State-of-the-art protection CFI (Abadi et al. 2005; Bletsch et al. 2011; Egele et al. 2012; Zhang and Sekar 2013) and ASLR (Backes 2014; Bhatkar et al. 2003, 2005; Hiser et al. 2012; Kil et al. 2006; Keromytis et al. 2012; Paleari et al. 2009; The PaX Team 2003b; Wartell et al. 2012; Bigelow et al. 2015; Davi et al. 2015; Giuffrida et al. 2012; Lu et al. 2016) raise the bar for the attacker to get the information of memory layout.

When the canary detection discovers the DSMAs, it is a malicious/unsafe event clearly. If the point is that attackers

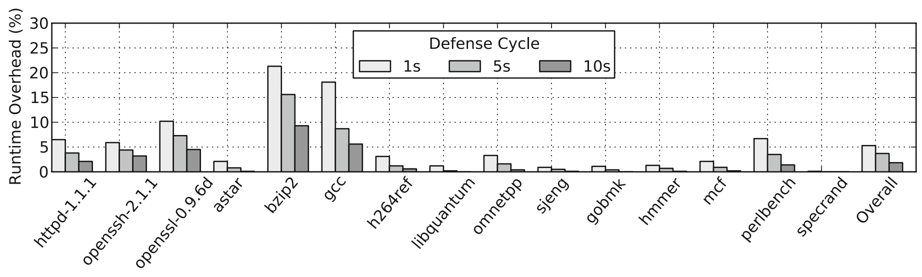

Fig. 2 Runtime overhead

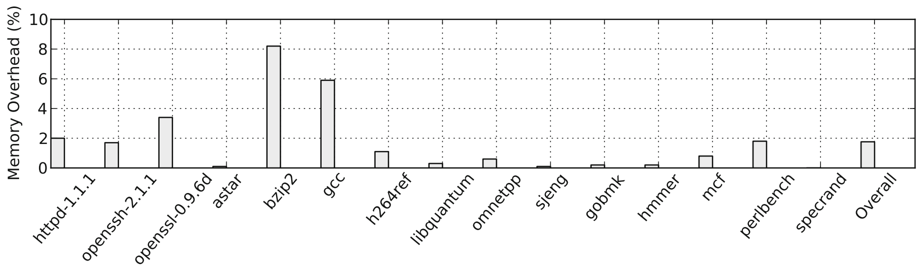

Fig. 3 Memory overhead

may continue poking around in a brute-force fashion, then blocking the attack after the detection of several consecutive crashes/canary overwrites seems to be a good idea. However, this kind of defense (stopping the process) actually has the same results under DDoS attacks. In addition, merely stopping cannot prevent the next but the same DSMAs.

Related work

In this section, we focus on two potential defenses against DSMAs: Data Flow Integrity and Data-Plane Randomization.

Data Flow Integrity DFI was first proposed by Miguel Castro et al. (2006). By using static analysis, DFI computes

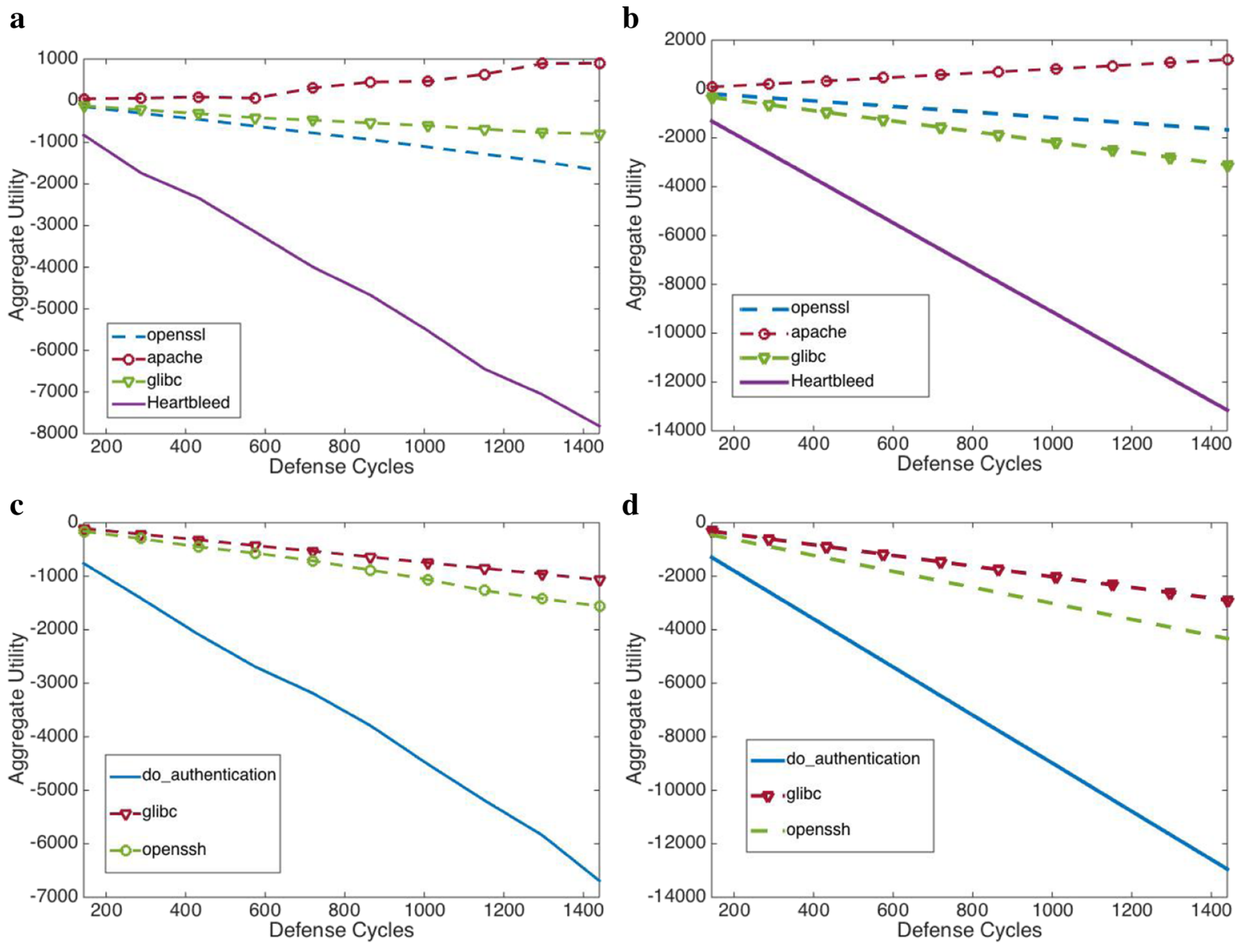

Fig. 4 a- **b**: Comparisons among the aggregate utility values in Matlab simulations and Apache experiments when defending against openssl (CVE-2002-0656), glibc (CVE-2015-0235), apache (CVE-1999-0071), and Heartbleed (CVE-2014-0610); **c** - **d**: the comparison among the aggregate utility values in Matlab simulations and Openssh experiments when defending against glibc (CVE-2015-0235), openssh (CVE-2001-0144), and do_authentication (CVE-2001-0144)

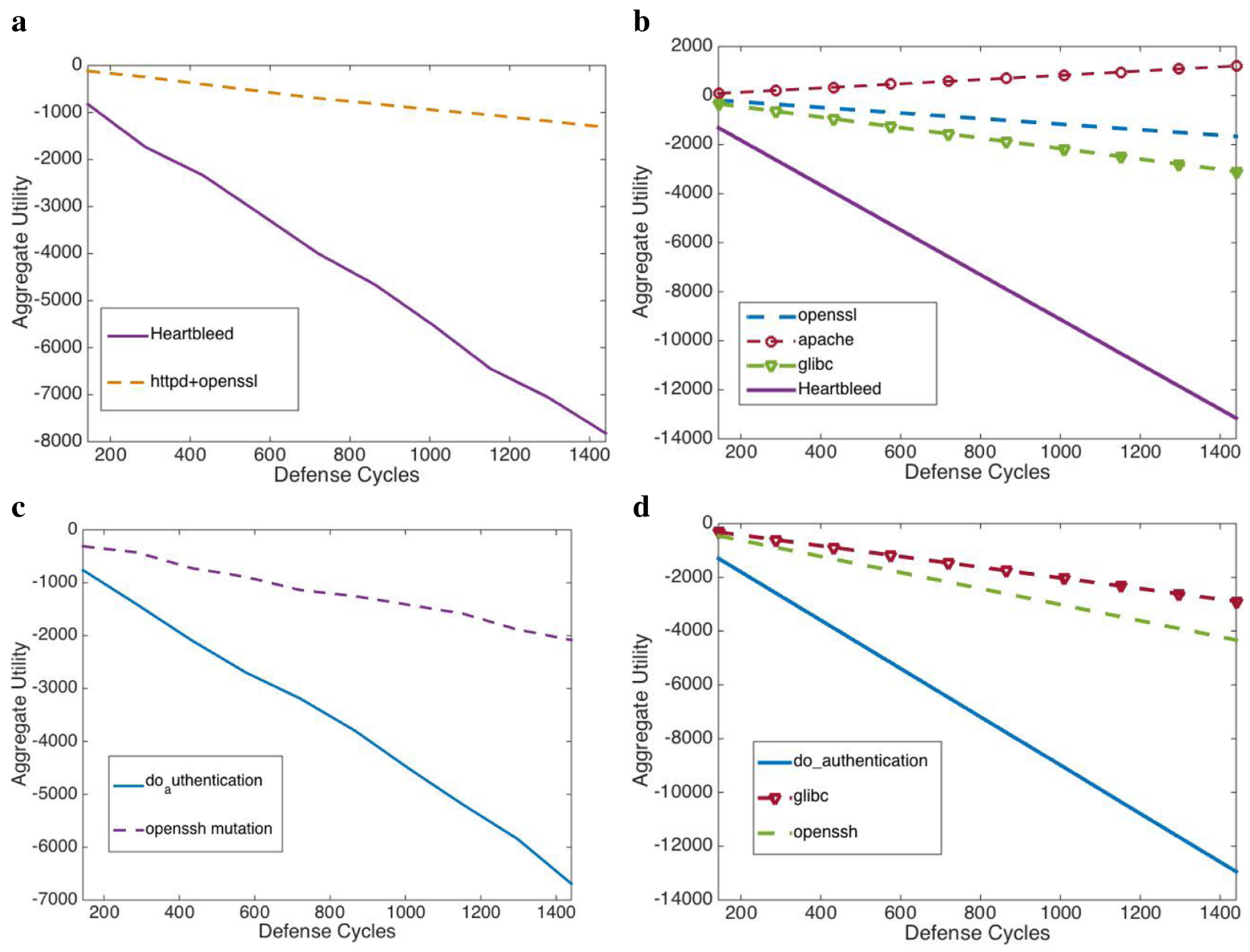

Fig. 5 a- **b** Compare the aggregate utility values of apache + openssl (CVE-2002-0656 & CVE-1999-0071), and Heartbleed (CVE-2014-0610) in real server experiments with the aggregate utility values in Matlab simulations; **c** - **d** compare the aggregate utility values of openssh mutation (CVE-2001-0144) and do_authentication (CVE-2001-0144) in real server experiments with the aggregate utility values in Matlab simulations

a Data Flow Graph and checks whether the definition of each data object is legal at run-time. A tailored DFI (Song et al. 2016) was proposed to solve the privilege escalation attack in the kernel. A complete enforcement of DFI can defend against DSMAs, however, complete DFI suffers from performance overhead as high as 103% (Castro et al. 2006). Tailored DFI (Song et al. 2016) focusing on privilege escalation attacks in the kernel can defeat small parts of DSMAs, but the majority of DSMAs are out-of-scope of that work. Recently, researchers leverage hardware to assist the DFI and improve the runtime overhead (Song et al. 2016). However, the techniques heavily depend on new features of the newest CPU (processor tracing), which is not in use by the majority of web service providers. By comparison, our method not only has reasonable performance overhead, but also is hardware independent.

Data-Plane Randomization Data Space Randomization (DSR) (Bhatkar and Sekar 2008; Cadar et al. 2008) was proposed to prevent non-control-flow attacks by XORing data with random masks. However DSR introduces high performance overhead since all the data objects need to be randomized. Static DSLR (Lin et al. 2009; Stanley et al. 2013; Xin et al. 2010) was proposed to prevent data structure manipulation attacks, via modifying the definition of a data structure to reorder the fields. However, static DSLR has several limitations. First, the layout randomized by static DSLR is determined at compile time, and it is vulnerable to brute force attacks (Stacham et al. 2004). Second, static DSLR requires manual efforts to determine which data structure can be randomized. Xin et al. (Xin et al. 2010) extend static DSLR and propose to use a constraint set to select randomizable data structures. But their technique cannot handle nested data structures and ignores all data structures associated with pointer operations. Recently, SALADS (Chen et al. 2015) was proposed to achieve dynamical data structure layout re-randomization. However, the set of randomized data structures selected by an expert is fixed throughout the whole lifetime of a process. In addition, SALADS suffers from high runtime overhead.

Conclusion

We present SALADSPlus, a new adaptive DSLR with adversarial reasoning, that automatically translates a program to a data structure self-randomizing (DSSR) program. At runtime, a DSSR program periodically selects and randomizes a set of data structures based on the UCB-D algorithm. Besides, SALADSPlus could perform adversary reasoning to indicate whether there is DSMA or not, and further locate the target program of the attack. SALADSPlus is the first effective defense with low overhead against DSMA. Moreover, adversarial reasoning is a unique feature of our defense. We have implemented SALADSPlus based on gcc-4.5.0. Experimental results show that the runtime overheads are 1.8%, 3.7%, and 5.3% when the defense cycles are selected as 10s, 5s, and 1s respectively.

Endnotes

[1] ASLR randomizes the base addresses of both data and code in the memory.

[2] CFI disables deviations from the being-protected program's original control-flow graph.

[3] We used brute force attacks to guess the layout of `ssl_session_st` when we did the experiments.

Acknowledgements
This work was supported by ARO W911NF-13-1-0421 (MURI), NSF CNS-1422594, and NSF CNS-1505664.

Authors' contributions
PC carried out the background, idea proposal, system implementation and evaluation, ZH designed the UCB-D algorithm, participated in the experiments and drafted the manuscript. JX carried out the improvements of the manuscripts. MZ participated in the problem formulation, designed the UCB-D algorithm and drafted the manuscript. PL conceived of the study and participated in its design and coordination. All authors read and approved the final manuscript.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Author details
[1]College of Information Sciences and Technology, The Pennsylvania State University, University Park 16802, PA, USA. [2]The School of Electrical Engineering and Computer Science, The Pennsylvania State University, State College, University Park 16801, PA, USA.

Received: 4 January 2018 Accepted: 17 April 2018
Published online: 05 June 2018

References

Abadi M, Budiu M, Erlingsson U, Ligatti J (2005) Control-flow integrity. In: ACM Conference on Computer and Communications Security (CCS '05). ACM, New York

Auer P, Cesa-Bianchi N, Fischer P (2002) Finite-time analysis of the multiarmed bandit problem. Mach Learn 47(2-3):235–256

Backes M, Nürnberger S (2014) Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In: USENIX Security Symposium (Security '14). USENIX Association, San Diego

Bhatkar E, Duvarney DC, Sekar R (2003) Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: USENIX Security Symposium (Security '03). USENIX Association, San Diego

Bhatkar S, Sekar R (2008) Data space randomization. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '08). Springer-Verlag, Berlin

Bhatkar S, Sekar R, DuVarney DC (2005) Efficient techniques for comprehensive protection from memory error exploits. In: USENIX Security Symposium (Security '05). USENIX Association, San Diego

Bigelow D, Hobson T, Rudd R, Streilein W, Okhravi H (2015) Timely rerandomization for mitigating memory disclosures. In: Proceedings of the 22nd Conference on Computer and Communications Security (CCS '15). ACM, New York

Bittau A, Belay A, Mashtizadeh A, Mazieres D, Boneh D (2014) Hacking blind. In: IEEE Symposium on Security and Privacy (Oakland '14). IEEE Computer Society, Washington

Blazakis D (2010) Interpreter exploitation. In: USENIX Conference on Offensive Technologies (WOOT '10). IEEE Computer Society, Washington

Bletsch T, Jiang X, Freeh V (2011) Mitigating code-reuse attacks with control-flow locking. In: Annual Computer Security Applications Conference (ACSAC '11). ACM, New York

Cadar C, Akritidis P, Costa M, Martin J-P, Castro M (2008) Data randomization. In: MSR-TR-2008-120. Microsoft Research, Cambridge

Castro M, Costa M, Harris T (2006) Securing software by enforcing data-flow integrity. In: Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI'06). USENIX Association, Berkeley

Chen H, Mao Y, Wang X, Zhou D, Zeldovich N, Kaashoek MF (2011) Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In: Asia-Pacific Workshop on Systems (APSys '11). ACM, New York

Chen P, Xu J, Lin Z, Xu D, Mao B, Liu P (2015) A practical approach for adaptive data structure layout randomization. In: Proceedings of the 20th European Symposium on Research in Computer Security (ESORICS'15). Springer, Switzerland

Chen S, Xu J, Sezer EC, Gauriar P, Iyer RK (2005) Non-control-data attacks are realistic threats. In: Proceedings of the 14th Conference on USENIX Security Symposium (Security '05). USENIX Association, San Diego

Crispin C, Calton P, Dave M, Heather H, Jonathan W, Peat B, Steve B, Aaron G, Perry W, Qian Z (1998) Stackguard: automatic adaptive detection and prevention of buffer-overflow attacks. In: USENIX Security Symposium (Security '98). USENIX Association, San Diego

CVE-1999-0071 (1999) Apache-cookie bug. http://seclab.cs.ucdavis.edu/projects/testing/vulner/39.html

CVE-2001-0144 (2001) SSH CRC-32 compensation attack detector. http://www.securityfocus.com/bid/2347/discuss

CVE-2002-0656 (2002) Apache openssl heap overflow exploit. http://www.phreedom.org/research/exploits/apache-openssl/

CVE-2014-0160 (2014) Heartbleed Bug

CVE-2015-0235 (2015) Ghost: glibc gethostbyname buffer overflow. https://www.qualys.com/2015/01/27/cve-2015-0235/GHOST-CVE-2015-0235.txt

Davi L, Liebchen C, Sadeghi A-R, Snow KZ, Monrose F (2015) Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In: Annual Network and Distributed System Security Symposium (NDSS '15). NDSS Symposium, San Diego

Egele M, Fischer T, Holz T, Hund R, Nurnberger S, Sadeghi AR, Davi L, Dmitrienko A (2012) Mocfi: A framework to mitigate control-flow attacks on smartphones,. In: Annual Network and Distributed System Security Symposium (NDSS'12). NDSS Symposium, San Diego

Giuffrida C, Kuijsten A, Tanenbaum AS (2012) Enhanced operating system security through efficient and fine-grained address space randomization. In: USENIX Conference on Security Symposium (Security '12). USENIX Association, San Diego
Hiser J, Nguyen-Tuong A, Co M, Hall M, Davidson JW (2012) Ilr: Where'd my gadgets go?. In: IEEE Symposium on Security and Privacy (Oakland '12). IEEE Computer Society, Washington
Hu H, Chua ZL, Adrian S, Saxena P, Liang Z (2015) Automatic generation of data-oriented exploits. In: Proceedings of the 24th USENIX Security Symposium (Security '15). USENIX Association, San Diego
Hu H, Shinde S, Adrian S, Chua ZL, Saxena P, Liang Z (2016) Data-oriented programming: On the expressiveness of non-control data attacks. In: IEEE Symposium on Security and Privacy (Oakland '16). IEEE Computer Society, Washington
Keromytis AD, Pappas V, Polychronakis M (2012) Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In: IEEE Symposium on Security and Privacy (Oakland '12). IEEE Computer Society, Washington
Kil C, Jim J, Bookholt C, Xu J, Ning P (2006) Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In: Annual Computer Security Applications Conference (ACSAC '06). IEEE, Miami Beach
Kuleshov V, Precup D (2014) Algorithms for multi-armed bandit problems. In: Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm, Society for Industrial and Applied Mathematics Philadelphia, PA, USA. pp 928–936. CVE-2014-0160 (2014) https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160
Lai TL, Robbins H (1985) Asymptotically efficient adaptive allocation rules. Adv Appl Math 6(1):4–22
Lin Z, Riley RD, Xu D (2009) Polymorphing software by randomizing data structure layout. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '09). Berlin, Springer-Verlag
Litchfield D (2003) Defeating the stack based buffer overflow prevention mechanism of microsoft windows 2003 server. https://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-litchfield.pdf
Lu K, Nurnberger S, Backes M, Lee W (2016) How to make aslr win the clone wars: Runtime re-randomization. In: Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16). NDSS Symposium, San Diego
Microsoft (2008) A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2. http://support.microsoft.com/kb/875352
OWASP (2009) Format string. https://www.owasp.org/index.php/Format_string_attack
Paleari R, Roglia GF, Martignoni L (2009) Surgically returning to randomized lib(c). In: Annual Computer Security Applications Conference (ACSAC '09). ACM, New York
Seibert J, Okhravi H, Söderström E (2014) Information leaks without memory disclosures:remote side channel attacks on diversified code. In: ACM Conference on Computer and Communications Security (CCS '14). ACM, New York
Shacham H, Page M, Pfaff B, Goh E-J, Modadugu N, Boneh D (2004) On the effectiveness of address-space randomization. In: ACM Conference on Computer and Communications Security (CCS '04). ACM, New York
Snow KZ, Monrose F, Davi L, Dmitrienko A, Liebchen C, Sadeghi A-R (2013) Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In: IEEE Symposium on Security and Privacy (Oakland '13). IEEE, Berkeley
Song C, Lee B, Lu K, Harris WR, Kim T, Lee W (2016) Enforcing kernel security invariants with data flow integrity. In: Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS '16). NDSS Symposium, San Diego
Song C, Moon H, Alam M, Yun I, Lee B, Kim T, Lee W, Paek Y (2016) Hdfi: Hardware-assisted data-flow isolation. In: Proceedings of IEEE Symposium on Security and Privacy (Oakland '16). NDSS Symposium, San Diego
Stanley DM, Xu D, Spafford EH (2013) Improved kernel security through memory layout randomization. In: International Performance Computing and Communications Conference (IPCCC '13). IEEE, San Diego
Team C (2009) Exploit writing tutorial part 6 : Bypassing stack cookies, safeseh, sehop, hw dep and aslr. https://www.corelan.be/index.php/2009/09/21/exploit-writing-tutorial-part-6-bypassing-stack-cookies-safeseh-hw-dep-and-aslr/
The PaX Team (2003a) PaX non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt
The PaX Team (2003b) Pax address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt
Wartell R, Mohan V, Hamlen K, Lin Z (2012) Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In: ACM Conference on Computer and Communications Security (CCS '12). ACM, New York
Xin Z, Chen H, Han H, Mao B, Xie L (2010) Misleading malware similarities analysis by automatic data structure obfuscation. In: International Conference on Information Security (ISC '10). Springer-Verlag, Berlin
Zhang Y, Juels A, Reiter MK, Ristenpart T (2012) Cross-vm side channels and their use to extract private keys. In: ACM Conference on Computer and Communications Security (CCS '12). ACM, New York
Zhang M, Sekar R (2013) Control flow integrity for cots binaries. In: USENIX Conference on Security (Security '13). USENIX Association, San Diego